Kata 1: Primeros pasos

Objetivo: Crear un repositorio local y hacer el primer commit.
Pasos:

1. Crear una carpeta y entrar en ella:

   ```
   mkdir kata-git && cd kata-git
   ```

2. Iniciar un repositorio Git:

   ```
   git init
   ```

3. Crear un archivo `hola.py` con contenido:

   ```
   print("Hola mundo")
   ```

4. Verificar el estado:

   ```
   git status  # Mostrará "hola.py" como "untracked" (sin
           seguimiento)
   ```

5. Agregar y hacer commit:

   ```
   git add hola.py
   git commit -m "Primer commit: saludo inicial"
   ```

   Pregunta:

- Antes del `add`, `git status` muestra archivos como “untracked”. Después del commit, aparecen en el historial (`git log`).

Kata 2: Segundo commit y exploración

Objetivo: Modificar un archivo y revisar cambios antes de commitear.
Pasos:

1. Editar `hola.py` para que pida un nombre:

   ```
   nombre = input("¿Cómo te llamas? ")
   print(f"Hola, {nombre}")
   ```

2. Ver diferencias con la versión anterior:

   ```
   git diff  # Muestra los cambios no agregados
   ```

3. Agregar y commitear:

```
git add hola.py
git commit -m "Añadido input de nombre"
```

Pregunta:

- `git diff` muestra cambios **antes** de add. Después de add, los cambios están en el "staging area" y ya no se ven con `git diff` (usa `git diff --cached`).

Kata 3: Volver al pasado

Objetivo: Navegar entre commits antiguos.
Pasos:

1. Ver historial de commits (con hash corto):

   ```
   git log --oneline
   ```

2. Ir a un commit anterior (usando su hash):

   ```
   git checkout <hash-del-commit>
   ```

3. Volver a `main`:

   ```
   git checkout main
   ```

Pregunta:

- Al estar en un commit antiguo, los archivos se **actualizan** a su estado en ese momento. Si haces cambios aquí, estarás en un estado "detached HEAD" (no recomendado para trabajo normal).

Kata 4: Branches para probar ideas

Objetivo: Trabajar con ramas.
Pasos:

1. Crear una rama `saludo-ingles`:

   ```
   git checkout -b saludo-ingles
   ```

2. Cambiar el mensaje en `hola.py`:

   ```
   print("Hello world")
   ```

3. Hacer commit:

   ```
   git commit -am "Saludo en inglés"
   ```

4. Volver a `main`:

   ```
   git checkout main
   ```

   Pregunta:

- Los cambios en `saludo-ingles` están **aislados** de `main` hasta que hagas merge.

Kata 5: Comparar ramas

Objetivo: Ver diferencias entre ramas.
Pasos:

1. Comparar `main` con `saludo-ingles`:

   ```
   git diff main saludo-ingles  # Muestra cambios de main ->
           saludo-ingles
   ```

2. Ver gráfico de ramas:

   ```
   git log --graph --oneline --all
   ```

   Pregunta:

- `git diff A B` muestra cambios de `A` a `B`. Si inviertes el orden, verás las diferencias al revés (líneas eliminadas vs. añadidas).

Kata 6: Fusionar cambios

Objetivo: Unir ramas con `merge`.
Pasos:

1. Desde `main`, fusionar `saludo-ingles`:

   ```
   git merge saludo-ingles
   ```

2. Ver historial:

   ```
   git log --oneline  # Mostrará un commit de merge
   ```

   Pregunta:

- Los commits de `saludo-ingles` ahora son parte del historial de `main`.

Kata 7: Manejo de conflictos

Objetivo: Resolver un conflicto manualmente.
Pasos:

1. Crear rama `saludo-frances`:

   ```
   git checkout -b saludo-frances
   ```

2. Cambiar `hola.py`:

   ```
   print("Bonjour")
   ```

3. Hacer commit y volver a `main`.

4. En `main`, cambiar `hola.py` a `print("Hallo")` y commitear.

5. Intentar merge:

   ```
   git merge saludo-frances  # ¡Conflicto!
   ```

6. Editar el archivo para resolver el conflicto (eliminar marcas `<<<<<<<`, `=======`, `>>>>>>>`), luego:

   ```
   git add hola.py
   git commit -m "Resuelto conflicto: saludo en alemán y
           francés"
   ```

   Pregunta:

- Git marca los conflictos en el archivo con símbolos especiales. Debes editarlo manualmente.

Kata 8: Deshacer cambios

Objetivo: Revertir commits con `reset`.
Pasos:

1. Hacer un commit no deseado (ej: borrar una línea).
2. Deshacerlo según el caso:

   - **Suave** (mantiene cambios en staging):

     ```
     git reset --soft HEAD~1
     ```

   - **Mixto** (mantiene cambios en working directory):

     ```
     git reset --mixed HEAD~1  # Opción por defecto
     ```

    - **Duro** (elimina todo):
      bash `git reset --hard HEAD~1  # ¡Cuidado! Pierdes cambios`. **Pregunta**:

- `--soft` es el más seguro (archivos intactos, cambios quedan en staging). `--hard` es destructivo.

Resumen visual de comandos clave:

Kata	Comandos principales
1	`git init`, `git add`, `git commit`
2	`git diff`, `git add`
3	`git log`, `git checkout <hash>`
4	`git branch`, `git checkout -b`
5	`git diff rama1 rama2`
6	`git merge`
7	`git merge`, editar conflicto
8	`git reset --soft/--mixed/--hard`

Kata 9: Ignorar archivos innecesarios con `.gitignore`

Objetivo: Evitar que Git rastree archivos irrelevantes o sensibles (como contraseñas o archivos temporales).

Pasos:

1. **Crear un archivo de configuración** (ejemplo con datos sensibles):

   ```
   echo "clave_secreta=12345" > config.txt
   ```

2. **Verificar el estado** (Git lo detectará como “untracked”):

```
git status
```

3. **Crear el archivo `.gitignore`** (en la raíz del proyecto):

   ```
   touch .gitignore
   ```

   - Edítalo con tu editor (VS Code, Nano, etc.) y añade:

     ```
     config.txt   # Ignora este archivo específico
     *.log        # Ignora TODOS los archivos .log
     /temp/       # Ignora la carpeta 'temp'
     ```

4. **Verificar que Git ignora los archivos**:

   ```
   git status  # config.txt ya no aparecerá (si no estaba
           agregado antes)
   ```

Si ya habías agregado `config.txt` por error:

```
git rm --cached config.txt  # Lo elimina del staging, pero lo
        mantiene en tu disco
git commit -m "Eliminado config.txt del seguimiento"
```

Prueba opcional:

- Crea archivos `.log` y verifica que Git los ignora:

  ```
  echo "Log de prueba" > registro.log
  git status  # No aparecerá
  ```

Preguntas de reflexión:

- **¿Por qué no subir `config.txt` o `.env` a GitHub?**
  → Porque pueden contener contraseñas o claves API, exponiendo tu proyecto a ataques.
- **¿Qué pasa si alguien sube un `.env` con contraseñas?**
  → Debe revocarlas inmediatamente y borrar el historial con `git filter-branch` o herramientas como BFG Repo-Cleaner.
- **¿Puedes tener varios `.gitignore`?**
  → Sí, en subdirectorios para reglas específicas (ej: `node_modules/` en proyectos Node.js).

Comandos clave:

```
git status
git rm --cached archivo
nano .gitignore  # Para editarlo
```

Kata 10: Publicar el repositorio en GitHub (Remoto)

Objetivo: Subir tu proyecto local a un repositorio remoto en GitHub.

Pasos:

1. **Crear un repositorio vacío en GitHub**:
   - Ve a github.com/new.
   - Dale un nombre (ej: `kata-git`) y haz clic en "Create repository".
2. **Conectar tu repositorio local al remoto**:

   ```
   git remote add origin https://github.com/tu-usuario/kata-git.git
   ```

   - Reemplaza `tu-usuario` con tu nombre de GitHub.
3. **Subir los cambios** (por primera vez):

   ```
   git push -u origin main
   ```

   - El flag `-u` establece `origin main` como la rama predeterminada para futuros `push`.

Pregunta:

- **¿Qué hace `-u` en `git push -u`?**
  → Guarda la asociación entre tu rama local (`main`) y la remota (`origin/main`). Después, puedes usar solo `git push`.

Comandos clave:

```
git remote -v             # Verifica la conexión remota
git push -u origin main   # Primer push
git push                  # Push posteriores (sin -u)
```

Resumen visual:

Kata	Concepto clave	Comandos útiles
9	Ignorar archivos	`.gitignore`, `git rm --cached`
10	Subir a GitHub	`git remote add`, `git push -u`

Consejos adicionales:

- **Para el `.gitignore`**: Usa plantillas predefinidas para tu lenguaje (ej: github/gitignore).
- **Si `git push` falla**: Asegúrate de tener permisos en el repositorio remoto y que la rama se llame `main` (no `master`).